MICROCHIP

USB Mass Storage Class on an Embedded Device

Author: *Sean Justice*
Microchip Technology Inc.

INTRODUCTION

In recent years, there has been immense growth in Universal Serial Bus (USB) based applications, primarily due to the Plug-and-Play nature of USB.

This application note discusses and provides a Mass Storage Device (MSD) function driver that can be integrated with almost any application running on Microchip 32-bit PIC® microcontroller products with USB peripheral support. It describes the design and implementation of a USB MSD function driver using a Secure Digital card, which is useful to developers of USB mass storage solutions.

This application may be used as a stand-alone MSD or as a Secure Digital/Multimedia Card (SD/MMC) reader/writer interface. The MSD function driver uses the Microchip PIC32 USB device stack.

ASSUMPTIONS

The author assumes that the reader is familiar with the following Microchip development tools: MPLAB® IDE and MPLAB REAL ICE™ in-circuit emulator. It is also assumed that the reader is familiar with C programming language, and USB device protocol and descriptors. Terminology from these technologies is used in this document, and only brief overviews of the concepts are provided. Advanced users are encouraged to read the associated specifications.

FEATURES

This application note provides key components of an MSD device function driver. The Microchip MSD device function driver incorporates the following features:

- Functions independently of RTOS or application
- Supports Microchip MPLAB IDE tool suite
- Supports the MSD 1.0 specification
- Uses two USB endpoints that are configured for bulk transfers
- File system support (FAT16, FAT32, NTFS) is dependent on the host OS
- No custom drivers required
- Handles standard MSD USB configuration requests, as stated in Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"* (available on the Internet at the following URL: http://www.usb.org/developers/docs/

LIMITATIONS

Since the MSD function driver is developed for use in embedded systems, the limitations are those that are inherited by the USB device stack (refer to Microchip Application Note AN1176, *"USB Device Stack for PIC32 Programmer's Guide"*.

SYSTEM HARDWARE

This application and firmware was developed for the following hardware:

- PIC32MX Family Microcontroller PIM (Processor Interface Module), supporting USB
- Microchip Explorer 16 Development Board
- USB PICtail™ Plus Daughter Board
- PICtail™ Daughter Board for SD and MMC Cards

The USB device and MSD function driver source files can be modified to use an alternative development board and accommodate most hardware differences.

PIC® MCU MEMORY RESOURCE REQUIREMENTS

The MSD function driver inherits all of the memory requires of the USB device stack. Refer to the **“Memory Resource Requirements”** section of AN1176*, “USB Device Stack for PIC32 Programmer’s Guide”* for more information.

The MSD function driver consumes Flash and RAM memory as shown in the following table:

TABLE 1: MEMORY REQUIREMENTS

Memory	Size
Flash	11,700 Bytes
RAM	608 Bytes

The MSD function driver application defines the following items:

- USB descriptor table
- MSD report structure

The USB descriptor table and MSD report structure, are required for any MSD function driver application.

USB descriptor table memory requirements are shown in the following table:

TABLE 2: USB DESCRIPTOR TABLE

Memory	Size
Flash	188 Bytes

The amount of memory resources consumed by the USB descriptor table may vary, depending on various factors, including, but not limited to, whether multiple USB function drivers are used. If more than one driver is used, the number of configurations, interfaces, and endpoint configurations will affect memory consumption.

INSTALLING SOURCE FILES

The complete source for the Microchip MSD function driver is available for download from the Microchip web site (see source code). The source code is distributed in a single Microsoft Windows® installation file.

Perform the following steps to complete the installation:

1. Execute the installation file. A Windows installation wizard will guide you through the installation process.
2. Before continuing with the installation process, you must accept the software license agreement by clicking **I accept**.
3. After completion of the installation process, you should see the following directory structure:
   a) `msd_device_driver` directory under `\PIC32 Solutions\Microchip\USB`. This directory contains the source files and documentation for the MSD function driver.
   b) `USB` directory under `\PIC32 Solutions\Microchip\Include`. This directory contains the include files for the MSD function driver.
   c) `usb_msd_device_demo` directory under `\PIC32 Solutions`. This directory contains the demo project and source files for the MSD function driver mouse demo.
4. Refer to the release notes for the latest version-specific features and limitations.

SOURCE FILE ORGANIZATION

The MSD device class consists of multiple files. These files are organized in multiple directories. Table 3 shows the directory structure.

TABLE 3: MSD SOURCE FILE DIRECTORY STRUCTURE

File	Directory	Description
`msd.c`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	USB MSD device class driver
`msddsc.tmpl`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	MSD descriptor template
`mediasd.c`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	SD interface API
`sdcard.tmpl`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	SD user modifiable defines template
`usb_device_msd.h`	`\PIC32 Solutions\Microchip\Include\USB`	API defines and modifiable macros
`msd_pri.h`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	Private function and macro defines
`mediasd.h`	`\PIC32 Solutions\Microchip\Include\USB`	SD interface header file

DEMO APPLICATION

An application that demonstrates the MSD function driver by simulating a removable disk drive is included with the Microchip MSD function driver. This application is designed to run on the Explorer 16 development board with Microchip USB device stack software. However, the application can be modified to support most boards.

The disk-drive-simulation demo application performs the following services:

- USB device enumeration for MSD function driver (seen as a removable disk drive on Windows Explorer)
- Write, read, edit, or delete files on the simulated drive
- File system support (FAT16, FAT32, NTFS) is dependent on the host OS
- See the capacity of the SD memory card by using the disk properties function of the PC

Programming the Demo Application

To program a target with the demo application, you must have access to an MPLAB REAL ICE in-circuit emulator. The following procedure assumes that you will be using MPLAB IDE. If not, please refer to your specific programmer's instructions.

1. Connect MPLAB REAL ICE in-circuit emulator to the Explorer 16 board or your target board.
2. Apply power to the target.
3. Launch MPLAB IDE.
4. Select the PIC32 device supporting USB of your choice (required only if you are importing a hex file previously built).
5. Enable MPLAB REAL ICE as a programmer.
6. Import the previously built hex file into MPLAB, if you wish to use it.
7. If you are rebuilding the hex file, open the project file and follow the build procedure to create the application hex file.
8. The demo application contains necessary configuration options required for the Explorer 16 board. If you are programming another type of board, make sure that you select the appropriate oscillator mode from the MPLAB IDE configuration settings menu.
9. Select the "Programmer" menu option in MPLAB IDE, and click "Select Programmer->6 REAL ICE."
10. When MPLAB IDE has detected the REAL ICE in-circuit emulator and the PIC MCU, select the "Programmer" menu option and click "Program" to program the device.
11. After a few seconds, the message "Programming successful" will be displayed. If it is not, check the board and MPLAB REAL ICE connections. Refer to MPLAB IDE and REAL ICE online help for further assistance.
12. Remove power from the board and disconnect the MPLAB REAL ICE cable from the target board.
13. Reapply power to the board and make sure that the LCD reads "PIC32 MSD Device". If it does not, check your programming steps and repeat, if necessary.

Building the Demo Application

The demo application included in this application note can be built using the Microchip C32 C compiler. If required, port the source to the compiler that you customarily use with Microchip microcontroller products.

This application note includes a predefined MSD project file for use with Microchip MPLAB IDE. The project was created using a PIC32MX device with USB. If a different device is used, the appropriate device must be selected through the MPLAB IDE menu command.

In addition, the demo application project uses additional include paths as defined in the "Build Options" of MPLAB IDE.

The following include paths are required:

- `.\`
- `..\Microchip\Include`
- `..\..\Microchip\Include`

Table 4 lists the source files that are necessary to build the demo application.

TABLE 4: DEMO APPLICATION PROJECT FILES

File	Directory	Description
`main.c`	`\PIC32 Solutions\usb_msd_device_demo`	Main demo source file
`msddsc.c`	`\PIC32 Solutions\usb_msd_device_demo`	MSD USB descriptors
`sdcard.def`	`\PIC32 Solutions\usb_msd_device_demo`	SD card user defines
`HardwareProfile.h`	`\PIC32 Solutions\usb_msd_device_demo`	Hardware defines for the PIC32MX
`usb_config.h`	`\PIC32 Solutions\usb_msd_device_demo`	USB specific defines for helper functions
`msd.c`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	USB MSD source file
`mediasd.c`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	SD card APIs
`msd_pri.h`	`\PIC32 Solutions\Microchip\USB\msd_device_driver`	Private function and macro definitions
`usb_device_msd.h`	`\PIC32 Solutions\Microchip\Include\USB`	USB MSD include file
`mediasd.h`	`\PIC32 Solutions\Microchip\Include\USB`	SD card API prototypes and defines
`usb_device.c`	`\PIC32 Solutions\Microchip\USB`	USB device APIs
`usb_hal.c`	`\PIC32 Solutions\Microchip\USB`	USB hardware APIs
`usb_hal_core.c`	`\PIC32 Solutions\Microchip\USB`	USB hardware core APIs
`usb.h`	`\PIC32 Solutions\Microchip\Include\USB`	USB defines and API prototypes
`usb_ch9.h`	`\PIC32 Solutions\Microchip\Include\USB`	USB defines and support, as in Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"*
`usb_common.h`	`\PIC32 Solutions\Microchip\Include\USB`	USB common defines
`usb_device.h`	`\PIC32 Solutions\Microchip\Include\USB`	USB device defines and API prototypes
`usb_hal.h`	`\PIC32 Solutions\Microchip\Include\USB`	USB hardware support
`mstimer.c`	`\PIC32 Solutions\Microchip\Common`	1 millisecond timer
`ex16lcd.c`	`\PIC32 Solutions\Microchip\Common`	Explorer 16 Development Board LCD
`mstimer.h`	`\PIC32 Solutions\Microchip\Include`	1 millisecond timer defines
`ex16lcd.h`	`\PIC32 Solutions\Microchip\Include`	Explorer 16 Development Board LCD defines

The following is a high-level procedure for building the demo application. This procedure assumes that you are familiar with MPLAB IDE and will be using MPLAB IDE to build the application. If not, refer to the instructions for your programmer to create and build the project.

1. Make sure that source files for the Microchip MSD function driver are installed. If not, please refer to **"Installing Source Files"** section.
2. Launch MPLAB IDE and open the project file.
3. Use MPLAB IDE menu commands to build the project. Note that the demo project is created to compile properly when the source files are located in the wizard-recommended directory structure. If you have moved or installed the source files to another location, you must recreate or modify existing project settings to build. See **"Building the Demo Application"** for more information.
4. The build process should finish successfully. If not, make sure that your MPLAB IDE and compiler are setup correctly.

Application-Specific USB Support

In using the Microchip PIC32 USB device firmware stack, the MSD demo implements the following application-specific tables.

1. USB Descriptor Table
2. Endpoint Configuration Table
3. Function Driver Table

THE USB DESCRIPTOR TABLE

Every USB device must provide a set of descriptors (data structures) that describe the device and provide details to the USB host about which class drivers to use. These descriptors are provided, and the information they contain, is clearly defined in Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"* and *"Universal Serial Bus Mass Storage Class, Bulk-Only Transport, Revision 1.0"*, available on the USB web site. Refer to these documents for complete details.

The USB device descriptors can be organized into three groups:

- Device
- Configuration
- Strings

The device descriptor identifies the type of device and gives the number of possible configurations.

The configuration descriptors describe the types of interfaces and endpoints used. This group also includes class-specific descriptors.

The string descriptors, although generally optional, provide user-readable information that the host may display.

Demo Application Descriptor Table

The descriptor table that is provided by the demo application is included in the source file `msddsc.c` and outlined in **Appendix C: "USB MSD Function Driver Descriptor Table Definitions"**.

The demo application descriptor table can be modified to add other interfaces or configurations. However, it is advisable that a thorough understanding of Chapter 9 of the *"Universal Serial Bus Specification, Revision 2.0"* and other applicable device function-driver-specific specifications is achieved before attempting to modify a descriptor table.

ENDPOINT CONFIGURATION TABLE

The endpoint configuration table is used by the USB device stack to properly configure all endpoints by interface and alternate setting as defined by the descriptor table. The table identifies which function driver will be used to service events that occur on each endpoint.

Each table entry contains the following information:

- Maximum packet size
- Configuration flags
- Configuration number
- Endpoint number
- Interface number
- Alternate setting
- Index in device function table for the endpoint handler

The endpoint configuration table for the demo application contains one entry because the MSD function driver only requires two endpoints (Bulk-In and Bulk-Out). The following table can be found in the source file `msddsc.c`. Microchip application note AN1176*, "USB Device Stack for PIC32 Programmer's Guide"* provides additional information about the endpoint configuration table.

EXAMPLE 1: ENDPOINT CONFIGURATION TABLE

```
const EP_CONFIG _EpConfigTlb[] =
{
    {
       MSD_EP_OUT_SIZE,                                     // max pack size
       USB_EP_TRANSMIT | USB_EP_HANDSHAKE | USB_EP_RECEIVE, // configure for Tx & Rx
                                                            // enable handshaking
        1,                                                  // configuration number
        1,                                                  // endpoint number
        0,                                                  // interface number
        0,                                                  // alternate setting
        0                                                   // handler function index

    },
};
```

FUNCTION DRIVER TABLE

Since a device may implement more then one class or vendor-specific USB device function driver, the Microchip USB device stack uses a table to manage access to support the function driver(s). Each table entry contains the information necessary to manage a single function driver.

Each table entry contains the following information:

- Initialization routine
- Event handler routine
- Initialization flags

The function driver table for the demo applications contains one entry because there is only one function driver: MSD. The following table can be found in the source file `msddsc.c`. Refer to AN1176*, "USB Device Stack for PIC32 Programmer's Guide"* for further information on the function driver table.

EXAMPLE 2: FUNCTION DRIVER TABLE

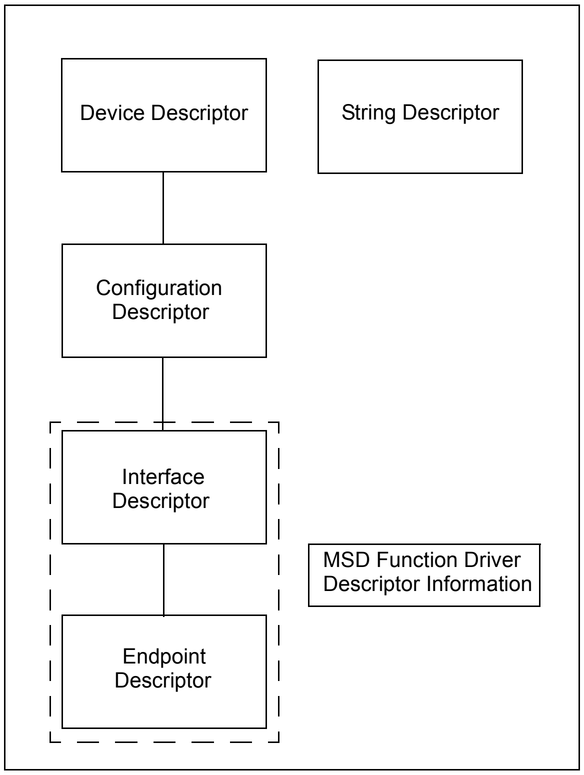
```
const FUNC_DRV _DevFuncTbl[] =
{
    {
        MSDInit,              // Initialization routine
        MSDEventhandler,      // Event handler routine
        0                     // Initialization flags
    }
};
```

MSD FUNCTION DRIVER OVERVIEW

MSD Descriptor

Every USB device has descriptor structure associated with it. A device may contain multiple function drivers e.g., HID, CDC, which are defined in the interface layer. Figure 1 shows a descriptor structure tree that would define an MSD class.

FIGURE 1: USB DESCRIPTOR FOR MSD FUNCTION DRIVER



MSD Command Block and Status Wrappers (CBW and CSW)

Bulk transfers are useful for transferring data when time is not a critical factor. Only high-speed and full-speed devices can do bulk transfers. A bulk transfer can send large amounts of data without overloading the bus, because it waits for the availability of the bus. The Mass Storage Class supports two transport protocols that determine which transfer type the device and host use to send command, data and status information. These two types of transport protocols are:

- Bulk-Only Transport (BOT)
- Control/Bulk/Interrupt (CBI) Transport

BOT is a data transport protocol that uses Bulk transport, whereas CBI transport uses Control transfer, Bulk transport and Interrupt transfer. In this application, BOT is used as the data transport protocol.

The Mass Storage Class specification defines two class-specific requests, `Get Max LUN` and `Mass Storage Reset`, that must be implemented by a mass storage device. (LUN [Logical Unit Number] is how a SCSI bus identified multiple drives [units] on a single bus.) Bulk-Only `Mass Storage Reset` is used to reset the mass storage device and its associated interface. The `Get Max LUN` request is used to determine the number of logical units that are supported by the device. The value of `Max LUN` can vary between 0 and 15 (1-16 logical devices). Note that the `LUN` starts from 0. The device may share multiple logical units that share the common device characteristics. The host should not send the Command Block Wrapper (CBW) to a non-existing `LUN`.

A device implementing BOT will support at least three endpoints: Control (required by all USB devices), Bulk-In and Bulk-Out. The USB 2.0 specification defines a control endpoint (Endpoint 0) as the default endpoint that does not require a descriptor. The Bulk-In endpoint is used for transferring data and status from the device to the host, and the Bulk-Out endpoint is used for transferring commands and data from the host to the device.

Note: USB defines data flow direction from the point of view of the host. An “IN” endpoint transmits data from the device to the host and an “OUT” endpoint receives data from the host.

Bulk-Only Transport (BOT)

Like a Control transfer, a Bulk Only Transfer (BOT) also consists of a Command stage, an optional Data stage and a Status stage. The Data stage may or may not be present for all command requests.

Figure 2 shows the flow of Command transport, Data-In, Data-Out and Status transport for BOT.

The Command Block Wrapper (CBW) is a short packet of exactly 31 bytes in length. The CBW and all subsequent data and Command Status Wrapper (CSW) start on a new packet boundary. It is important to note that all CBW transfers are ordered little-endian with LSB (byte 0) first.

FIGURE 2: COMMAND/DATA/STATUS FLOW IN BULK-ONLY TRANSPORT

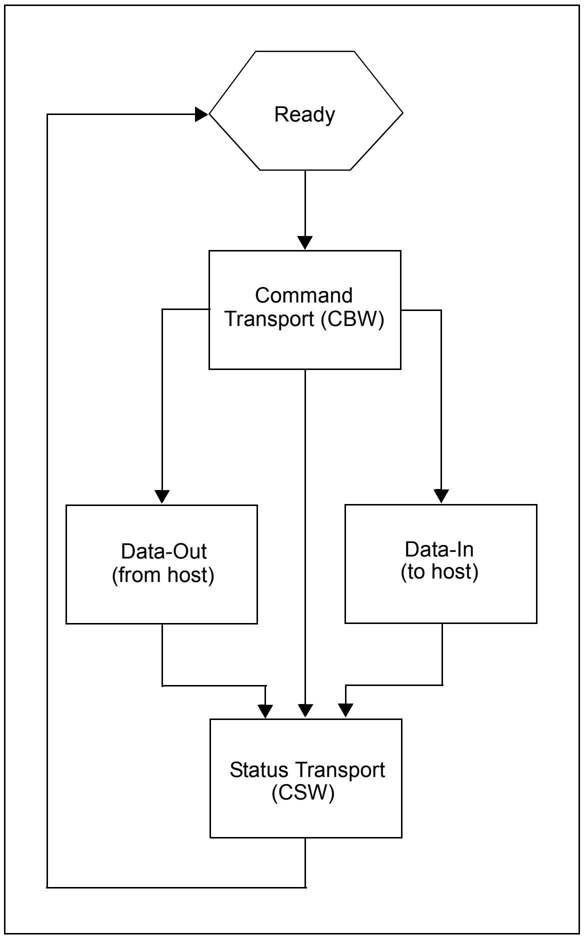


Table 5 shows the format of a CBW packet. In the CBW, the `dCBWSignature` value, “43425355h” (little-endian representation of the ASCII string “USBC”, for USB Command), identifies a CBW packet.

`dCBWTag` is the command block tag that is echoed back in CSW to associate the CSW with the corresponding CBW.

`dCBWDataTransferLength` indicates the number of bytes the host expects to transfer on a Bulk-In or Bulk-Out endpoint (as indicated by the Direction bit).

Only bit 7 of `bmCBWFlags` is used to indicate the direction of data flow, with a ‘`1`’ signifying Data-In (i.e., from device to host).

The field, `bCBWLUN`, specifies the device LUN to which the command block is being sent.

The field, `bCBWCB`, defines the valid length of the command block.

The `CBWCB` is the command block to be executed by the device.

TABLE 5: COMMAND BLOCK WRAPPER (CBW) FORMAT

Command Block Wrapper								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	dCBWSignature							
1								
2								
3								
4	dCBWTag							
5								
6								
7								
8	dCBWTransferLength							
9								
10								
11								
12	bmCBWFlags							
13	Reserved (0)				bCBWLUN			
14	Reserved (0)				bCDMCLength			
15...	CBWCB							
...30								

Table 6 shows the format of a CSW which is 13 bytes in length.

A `dCSWSignature` value of “53425355h” (little-endian representation of the ASCII string “USBS”, for USB Status) identifies a CSW packet.

The field, `dCSWTag`, echoes the `dCSWTag` value from the associated CBW.

For Data-Out, `dCSWDataResidue` is the difference between the data expected and the actual amount of data processed by the device.

For Data-In, it is the difference between the data expected and the actual amount of relevant data sent by the device.

The value of `dCSWDataResidue` is always less than or equal to the value of `dCBWDataTransferLength`.

The value of `bCSWStatus` indicates the success or failure of the command.

The `bCSWStatus` value of 00h indicates command success; 01h indicates command failure; and 02h indicates phase error.

TABLE 6: COMMAND STATUS WRAPPER (CSW) FORMAT

Command Status Wrapper								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	dCSWSignature							
1								
2								
3								
4	dCSWTag							
5								
6								
7								
8	dCSWDataResidue							
9								
10								
11								
12	bmCSWStatus							

SCSI Commands

After the successful enumeration of the target USB device, the host initiates commands according to the Interface Sub-Class specified in the interface descriptor during the enumeration process.

The USB MSD application specifies interface subclass of 06h, indicating that the device will support SCSI Primary Commands-2 (SPC-2) or later. An Interface Protocol value of 50h in the interface descriptor indicates that the BOT protocol is being used. As shown in Figure 2, a BOT transfer begins with a CBW. The device indicates the successful transport of a CBW by accepting (ACKing) the CBW. If the host detects a STALL of the Bulk-Out endpoint during Command transport, the host will respond with a Reset recovery. The host will attempt to transfer an exact number of bytes to or from the device as specified by the `dCBWDataTransferLength` and the Direction bit. The device will send each CSW to the host via the Bulk-In endpoint.

In this section, we briefly describe the SCSI commands that are supported in the MSD implementation. The reader may refer to SCSI Primary Commands-3 (SPC-3) and SCSI Block Commands-2 (SBC-2) specifications for further details. The first byte of the command block, `CBWCB`, is always the operation code or opcode in short.

- `INQUIRY` (Opcode 12h)
- `READ CAPACITY` (Opcode 25h)
- `READ (10)` (Opcode 28h)
- `WRITE (10)` (Opcode 2Ah)
- `REQUEST SENSE (6)` (Opcode 03h)
- `MODE SENSE (6)` (Opcode 1Ah)
- `PREVENT ALLOW MEDIUM REMOVAL` (Opcode 1Eh)
- `TEST UNIT READY` (Opcode 00h)
- `VERIFY (10)` (Opcode 2Fh)
- `START/STOP` (Opcode 1Bh)

The following items describe the operations:

`INQUIRY` (Opcode 12h):

The `INQUIRY` command requests that the information regarding the logical unit and SCSI target device is sent to the application client (host). The SPC-3 specification requires that the `INQUIRY` data should be returned, even though the device server is not ready for other commands. Moreover, the standard `INQUIRY` data should be available without incurring any media access delays. The standard `INQUIRY` data is at least 36 bytes and is shown in **Appendix D: “Standard Inquiry Format”**.

`READ CAPACITY` (Opcode 25h):

The `READ CAPACITY` command requests that the device server transfer bytes of parameter data that describe the capacity and medium format to the Data-In buffer. The response to the `READ CAPACITY` command is 4 bytes of returned `Logical Block Address` and 4 bytes of block length in bytes. Returned `Logical Block Address` is the logical block address of the last logical block on the direct access block device. If the number of logical blocks exceeds the maximum value that can be specified in the returned `Logical Block Address` field, the device will set the returned `Logical Block Address` field to FFFFFFFFh.

`READ (10)` (Opcode 28h):

The `READ (10)` command specifies that the device server reads the specified logical block(s) and transfers them to the Data-In buffer. The `READ (10)` command is a 10-byte `CBWCB`, with the eighth and ninth bytes specifying the `TRANSFER LENGTH`. The `TRANSFER LENGTH` field specifies the number of contiguous logical blocks of data that will be read and transferred to the Data-In buffer, starting with the logical block specified by the `Logical Block Address` field (bytes 3-6). A `TRANSFER LENGTH` field set to zero specifies that no logical blocks will be read.

`WRITE (10)` (Opcode 2Ah):

The `WRITE (10)` command requests that the device server transfers the specified logical blocks from the Data-Out buffer and write them. The `CBWCB` format for the `WRITE (10)` command is the same as the `READ (10)` command with `TRANSFER LENGTH` specifying the number of contiguous logical blocks of data that will be transferred from the Data-Out buffer and written, starting with the logical block specified by the `Logical Block Address` field. A `TRANSFER LENGTH` field set to zero specifies that no logical blocks will be written.

REQUEST SENSE (6) (Opcode 03h):

The REQUEST SENSE (6) command requests that the device server transfers the sense data to the application client. **Appendix E: "Fixed Format Sense Data"** shows the fixed format sense data response. The contents of the RESPONSE CODE field indicate the error type and format of the sense data. RESPONSE CODE 70h signifies the current error. RESPONSE CODE 71h signifies the deferred error in the fixed format sense data. And, code values 72h and 73h indicate the current and deferred error code in the descriptor format sense data. The SENSE KEY, ADDITIONAL SENSE CODE (ASC), and ADDITIONAL SENSE CODE QUALIFIER (ASCQ) fields provide a hierarchy of information. The SENSE KEY field indicates the generic information describing an error or exception condition. The ASC field indicates further information related to the error reported in the SENSE KEY field. The ASCQ field indicates the detailed information related to the ADDITIONAL SENSE CODE. Refer to Table 27 and Table 28 of the SPC-3 specification for a list of SENSE KEY error codes, and ASC and ASCQ error code assignments. This application implements the fixed format, current error code sense data response.

MODE SENSE (6) (Opcode 1Ah):

The MODE SENSE (6) command provides a means for a device server to report parameters to an application client. It is a complementary command to the MODE SELECT (6) command. The mode parameter header that is used by the MODE SENSE (6) and the MODE SELECT (6) command is shown in **Appendix F: "Mode Sense Header"**. The MEDIUM TYPE and DEVICE SPECIFIC PARAMETER fields are unique for each device type. In this application, the MEDIUM TYPE field is set to 00h, indicating a direct access block device. This value is the same as the value of the PERIPHERAL DEVICE TYPE field in the standard INQUIRY data.

TEST UNIT READY (Opcode 00h):

The TEST UNIT READY command provides a means to check if the logical unit is ready. This is not a request for self-check. If the logical unit is able to accept an appropriate medium access command, without returning a Check Condition status, this command will return a Good status. Otherwise, the command is terminated with the Check Condition status and the SENSE KEY is set to reflect the error condition.

PREVENT ALLOW MEDIUM REMOVAL (Opcode 1Eh):

The PREVENT ALLOW MEDIUM REMOVAL command requests that the logical unit enables or disables the removal of the medium. The prevention of medium removal will begin when an application client issues a PREVENT ALLOW MEDIUM REMOVAL command with a PREVENT field (fifth byte, bits 0-1 of CBWCB) of 01b or 11b (i.e., medium removal prevented). Because there is no way to prevent the removal of an SD card, the firmware decodes the command and prepares to notify the host PC that the operation has been successfully completed. If the medium is inaccessible, the command is specified as Fail (bCSWStatus = 0x01) with the SENSE KEY set to Not Ready.

VERIFY (10) (Opcode 2Fh):

The VERIFY (10) command requests that the device server verifies the specified logical block(s) on the medium. If the BYTCHK bit is set to '0', the device will perform medium verification with no data comparison, and will not transfer any data from the Data-Out buffer. If the BYTCHK bit is set to '1', the device server will perform a byte-by-byte comparison of user data that is read from the medium and user data that is transferred from the Data-Out buffer. The firmware decodes the command and then prepares to notify the host PC that the command has been successfully completed. If the medium is inaccessible, the command is specified as Fail, with the SENSE KEY set to Not Ready.

START/STOP (Opcode 1Bh):

The START/STOP command requests that the device server changes the power condition of the logical unit, load, or eject, the medium. This includes specifying that the device server enable or disable the direct access block device for medium access operations by controlling power conditions and timers. The POWER CONDITION field (fifth byte, bit 7-4) is used to specify that the logical unit be placed into a power condition or to adjust a timer. If the value of this field is not equal to 0h, the START (fifth byte, bit 0) and LOEJ (Load Eject, fifth byte, bit 1) bits are ignored. If the POWER CONDITION field is Active (1h), Idle (2h), or Standby (3h), the logical unit will transition to the specified power. If POWER CONDITION = 0h (START_VALID), the START, LOEJ = (0,0) signifies that the logical unit will transition to the stopped power condition; START, LOEJ = (0,1) signifies the logical unit will unload the medium; START, LOEJ = (1,0) signifies that the logical unit will transition to the active power condition. If START, LOEJ = (1,1), then the logical unit will load the medium.

UNSUPPORTED Commands

If the command opcode field in the `CBWCB` is not supported, the `SENSE KEY` is set to Illegal Request, indicating that there was an illegal parameter in the CDB, with `ASC` and `ACSQ` codes set corresponding to an invalid command opcode.

Secure Digital (SD) Card

A Secure Digital card is a common storage media used in portable devices (e.g., PDAs, digital cameras and MP3 players, among others). SD cards can be purchased in various storage sizes. Both SD cards and the MMC support the SPI transfer protocol, and have an electrical interface that is almost identical. While the form factor and the shape of the SD card and the MMC are identical, SD cards can be operated up to four times faster, have a write-protect switch, and may include cryptographic security for protection of copyrighted data. These features have increased the popularity of SD cards over MMC. Consequently, SD cards are the focus of this design. However, MMC have been tested and found to be fully functional with the MSD design.

An SD card can be operated in SD Bus mode or Serial Peripheral Interface (SPI) mode. In this application, the SD card is connected to the SPI bus of the PIC32MX and operated in that mode. In the SPI mode, only one data line is used for data transmission in each direction. Therefore, the data transfer rate in this mode is the same as it is in SD Bus mode with one data line (up to 25 Kbits per second).

Apart from the Power and Ground, the SPI bus consists of Chip Select ($\overline{\text{CS}}$), Serial Data Input (SDI), Serial Data Output (SDO) and Serial Clock (SCLK) signals. The SD card and MMC sample data input on the rising clock edge and set data output on the falling clock edge. On power-up, an SD card wakes up in SD Bus mode. Therefore, an initialization routine is required to operate the SD card in SPI mode. Initialization can be achieved by asserting the CS signal (logic low) during the reception of the Reset command, `CMD0`. In SPI mode, unlike SD Bus mode, the selected card always responds to the command. In case of a data retrieval problem, the card responds with an error response instead of a timeout, as it does in SD Bus mode. See **"References"** for information on the SD card specification.

Note:	Use of the information in these last two paragraphs may require a license from the SD Card Association.

CONCLUSION

This document and the associated custom demo application consisting of a mass storage demo using the MSD function driver.

Normally, managing the Universal Serial Bus requires that a developer handle protocols for device identification, control, and data transfer. However, Microchip has taken care of the USB details and provided a simple HID function driver to make implementing applications simple for developers who use supported Microchip microcontrollers.

REFERENCES

- Microchip Application Note AN1176*, “USB Device Stack for PIC32 Programmer’s Guide”* http://www.microchip.com
- *“Universal Serial Bus Specification, Revision 2.0”* http://www.usb.org/developers/docs
- Directory Snoop™, Version 5.01, http://www.briggsoft.com/dsnoop.htm
- FAT File System Specification – available by license, http://www.microsoft.com/mscorp/ip/tech/fat.asp
- MC74VHCT125A Data Sheet, http://www.onsemi.com
- MMC Specifications – some are available by license and others are available for purchase, http://www.mmca.org/compliance
- Microchip MPLAB® IDE In-circuit Development Environment, available free of charge, by license, from www.microchip.com/mplabide
- SCSI Primary Commands-2 (SPC-2), Revision i23, 18 July 2003, http://www.t10.org/ftp/t10/drafts/spc2/spc2i23.pdf
- SCSI Primary Commands-3 (SPC-3), Revision 21d, 14 February 2005, http://www.t10.org/ftp/t10/drafts/spc3/spc3r23.pdf
- SCSI Block Commands-2 (SBC-2), Revision 16, 13 November 2004, http://www.t10.org/ftp/t10/drafts/sbc2/sbc2r16.pdf
- SD Card Specification – available by license, http://www.sdcard.org
- SnoopyPro 0.22, http://sourceforge.net/projects/usbsnoop/
- *“USB Complete: Everything You Need to Develop Custom USB Peripherals”* by Jan Axelson, ISBN 0-9650819-5-8
- *“Universal Serial Bus Mass Storage Class Bulk-Only Transport, Revision 1.0”* http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf

APPENDIX A: MICROCHIP MSD FUNCTION DRIVER DEPENDENCIES

`MSDEventHandler` – USB Device Event Handler Function

The Microchip MSD function driver application note provides an event handler that is compliant with Microchip's AN1176, *"USB Device Stack for PIC32 Programmer's Guide"* application note. This routine must be placed into the defined USB device function driver table.

Syntax

```
PUBLIC BOOL MSDEventHandler(USB_EVENT event, void *data, UINT size)
```

Parameters

`event` – Enumerated data type identifying the event that has occurred (see below pre-defined events)

`data` – A pointer to event-dependent data (if available, see below)

`size` – The size of the event-dependent data, in bytes.

Return Values

`TRUE` if successful; `FALSE`, if not

Remarks

Refer to AN1176*, "USB Device Stack for PIC32 Programmer's Guide"* for more information.

`MSDInit` – USB Device Initialization Handler Function

This routine initializes all data structures associated with MSD Function Driver. This routine must be placed into the defined USB device function driver table.

Syntax

```
PUBLIC BOOL MSDInit(unsigned long flags)
```

Parameters

`flags` – reserved, pass a 0

Return Values

`TRUE` if initialization passed, else `FALSE`

Remarks

The `flags` parameter is passed as the initialization flags parameter of the USB device function driver table.

Note: Refer to Application Note AN1176*, "USB Device Stack for PIC32 Programmer's Guide"* for more information on `HIDEventHandler` and `HIDInit`.

APPENDIX B: MSD FUNCTION DRIVER MACROS

The USB Mass Storage Class on an Embedded Device provides several function driver macros to customize it for an application.

The following MSD function driver macros are available in the Microchip USB MSD Class on an Embedded Device and should be defined by the application in `usb_config.h`.

- `MSD_USB_ENDPNT`
- `MSD_EP_IN_SIZE`
- `MSD_EP_OUT_SIZE`

`MSD_USB_ENDPNT`

Purpose:	The endpoint from which the MSD function driver will send and receive data
Default:	1

`MSD_EP_IN_SIZE`

Purpose:	The size, in bytes, of the MSD function driver endpoint IN
Default:	32

`MSD_EP_OUT_SIZE`

Purpose:	The size, in bytes, of the MSD function driver endpoint OUT
Default:	64

APPENDIX C: USB MSD FUNCTION DRIVER DESCRIPTOR TABLE DEFINITIONS

TABLE C-1: DEVICE DESCRIPTOR

Field	MSD Function Driver Value	Description
`bLength`	12h	Size of this descriptor
`bDescriptorType`	1	Type, always USB device descriptor
`bcdUSB`	200h	USB specification version in BCD
`bDeviceClass`	0	Device class code
`bDeviceSubClass`	0	Device sub-class code
`bDeviceProtocol`	0	Device protocol
`bMaxPacketSize`	10h	Endpoint 0 max packet size
`idVendor`	4D8h	Vendor ID (VID)
`idProduct`	9	Product ID (PID)
`bcdDevice`	1	Device release number in BCD
`iManufacturer`	1	Manufacturer name string index
`iProduct`	2	Product description string index
`iSerialNum`	0	Product serial number string index
`bNumConfigurations`	1	Number of supported configurations

TABLE C-2: CONFIGURATION DESCRIPTOR

Field	MSD Function Driver Value	Description
`bLength`	9	Size of this descriptor
`bDescriptorType`	2	Type, always USB configuration descriptor
`wTotalLength`	20h	Total length of all descriptors
`bNumInterfaces`	1	Number of interfaces
`bConfigurationValue`	1	ID value
`iConfiguration`	0	Index of the string descriptor
`bmAttributes`		
`reserved_zero`	0	Always 0
`remote_waking`	1	1 if device supports remote wake-up
`self_powered`	0	1 if device is self-powered
`reserved_one`	1	Always 1
`bMaxPower`	50	milliamps/2 ex. 100 ma = 50

TABLE C-3: INTERFACE DESCRIPTOR

Field	MSD Function Driver Value	Description
bLength	9	Size of this descriptor
bDescriptorType	4	Type, always USB interface descriptor
bInterfaceNumber	0	ID number of the interface
bAlternateSetting	0	ID number of the alternate interface setting
bNumEndpoints	2	Number of endpoints in this interface
bInterfaceClass	8	USB Mass Storage interface class ID
bInterfaceSubClass	6	USB Mass Storage interface sub-class ID
bInterfaceProtocol	50h	USB SCSI transport interface BOT protocol ID
iInterface	0	Interface description string index

TABLE C-4: ENDPOINT DESCRIPTOR

Field	MSD Function Driver Value	Description
bLength	7	Size of this descriptor
bDescriptorType	5	Type, always USB endpoint descriptor
bEndpointAddress		
ep_num	1	Endpoint number
reserved	0	Always 0
direction	1	IN or OUT
bmAttribute		
transfer_type	0	Transfer type
synch_type	1	Synch type
usage_type	0	Usage type
reserved	0	Always 0
wMaxPacketSize	64	Largest packet the endpoint can handle
bInterval	0	Time between polling this endpoint for data

TABLE C-5: SERIAL STRING DESCRIPTOR

Field	MSD Function Driver Value	Description
bLength	4	Size of this descriptor
bDescriptorType	3	Type, always USB string descriptor
wLangid[]	409h	String

TABLE C-6: MANUFACTURE STRING DESCRIPTOR

Field	MSD Function Driver Value	Description
`bLength`	34h	Size of this descriptor
`bDescriptorType`	3	Type, always USB string descriptor
`wLangid[]`	Microchip Technology Inc.	String

TABLE C-7: SERIAL STRING DESCRIPTOR

Field	MSD Function Driver Value	Description
`bLength`	3Ah	Size of this descriptor
`bDescriptorType`	3	Type, always USB string descriptor
`wLangid[]`	Microchip Mass Storage Drive	String

APPENDIX D: STANDARD INQUIRY FORMAT

Standard Inquiry Format								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Peripheral Qualifiers			Peripheral Device Type				
1	RMB	Reserved						
2	Version							
3	Obsolete		NORMACA	HISUP	Response Data Format			
4	Additional Length (n-4)							
5	SCCS	ACC	TPGS		3PC	Reserved		PROTECT
6	BQUE	ENSERV	VS	MULTIP	MCHNGR	Obsolete		ADDR16
7	Obsolete		WUSB16	SYNC	LINKED	Obsolete	CMDQUE	VS
8	T10 Vendor Identification							
9								
10								
11								
12								
13								
14								
15								
16	Product Identification							
17								
18								
19								
20								
21								
22								
23								
24								
25								
26								
27								
28								
29								
30								
31								
32	Product Revision Level							
33								
34								
35								

APPENDIX E: FIXED FORMAT SENSE DATA

Fixed Format Sense Data								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Response Code (70h or 71h)							
1	Obsolete							
2	Filemark	EOM	ILI	Reserved	Sense Key			
3	Information							
4								
5								
6								
7	Additional Sense Length (n-7)							
8	Command Specific Information							
9								
10								
11								
12	Additional Sense Code							
13	Additional Sense Code Qualifiers							
14	Field Replacement Unit Code							
15	Sense Key Specific							
16								
17								
18...	Additional Sense Bytes							
...n								

APPENDIX F: MODE SENSE HEADER

Mode Sense Header								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	Mode Data Length							
1	Medium Type							
2	Device Specific Parameter							
3	Block Descriptor Length							

APPENDIX G: SOURCE CODE FOR THE MASS STORAGE CLASS FUNCTION DRIVER

The complete source code for the Microchip USB Mass Storage Class on an Embedded Device driver is offered under a no-cost license agreement. It is available for download as a single archive file from the Microchip corporate web site, at:

www.microchip.com.

After downloading the archive, always check the file release notes for the current revision level and a history of changes to the software.

REVISION HISTORY

Rev. A Document (02/2008)

This is the initial released version of this document.

NOTES:

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
== ISO/TS 16949:2002 ==

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

MICROCHIP

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08